inst.eecs.berkeley.edu/~cs61c/su05

CS61C : Machine Structures

Lecture #5: Memory Management

2005-06-27
Andy Carle

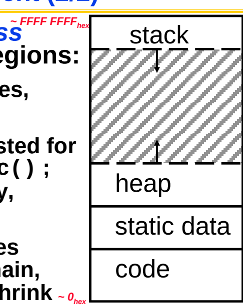
Memory Management (1/2)

- Variable declaration allocates memory
  - outside a procedure -> static storage
  - inside procedure -> stack
    - freed when procedure returns.
- Malloc request
  - Pointer: static or stack
  - Content: on heap

```
int myGlobal;
main() {
 int myTemp;
 int *f=
   malloc(16);
}
```


Memory Management (2/2)

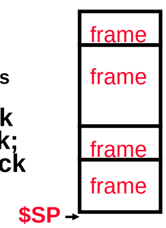
- A program's *address space* contains 4 regions:
  - stack: local variables, grows downward
  - heap: space requested for pointers via `malloc()` ; resizes dynamically, grows upward
  - static data: variables declared outside main, does not grow or shrink
  - code: loaded when program starts, does not change



For now, OS somehow prevents accesses between stack and heap (gray hash lines). Wait for virtual memory

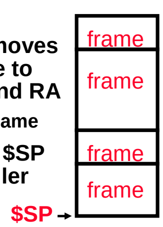
The Stack (1/4)

- Terminology:
  - Stack is composed of frames
  - A frame corresponds to one procedure invocation
  - Stack frame includes:
    - Return address of caller
    - Space for other local variables
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames



The Stack (2/4)

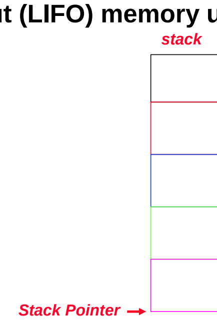
- Implementation:
  - By convention, stack grows down in memory.
  - Stack pointer ($SP) points to next available address
  - PUSH: On invocation, callee moves $SP down to create new frame to hold callee's local variables and RA
    - (old SP – new SP) ➔ size of frame
  - POP: On return, callee moves $SP back to original, returns to caller



The Stack (3/4)

- Last In, First Out (LIFO) memory usage

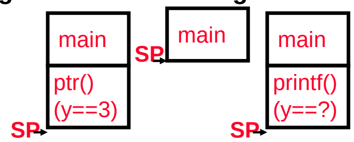
```
main ()
{ a(0);
}
  void a (int m)
  { b(1);
  }
    void b (int n)
    { c(2);
    }
      void c (int o)
      { d(3);
      }void d (int p)
        {
        }
```



The Stack (4/4): Dangling Pointers

- Pointers in C allow access to deallocated memory, leading to hard-to-find bugs !

```
int *ptr () {
   int y;
   y = 3;
   return &y;
}
main () {
   int *stackAddr;
   stackAddr = ptr();
   printf("%d", *stackAddr); /* 3 */
   printf("%d", *stackAddr); /* XXX */
}
```



Static and Code Segments

- Code (Text Segment)
  - Holds instructions to be executed
  - Constant size
- Static Segment
  - Holds global variables whose addresses are known at compile time
    - Compare to the heap (malloc calls) where address isn't known

The Heap (Dynamic memory)

- Large pool of memory, not allocated in contiguous order
  - back-to-back requests for heap memory could return blocks very far apart
  - where Java `new` command allocates memory
- In C, specify number of bytes of memory explicitly to allocate item

```
int *ptr;
ptr = (int *) malloc(4);
/* malloc returns type (void *),
so need to cast to right type */
```

  - `malloc()`: Allocates raw, uninitialized memory from heap

Memory Management

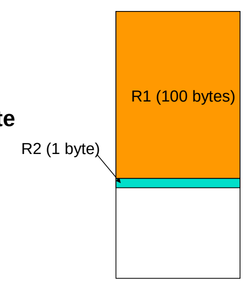
- How do we manage memory?
- Code, Static storage are easy: they never grow or shrink
- Stack space is also easy: stack frames are created and destroyed in last-in, first-out (LIFO) order
- Managing the heap is tricky: memory can be allocated / deallocated at any time

Heap Management Requirements

- Want `malloc()` and `free()` to run quickly.
- Want minimal memory overhead
- Want to avoid *fragmentation* – when most of our free memory is in many small chunks
  - In this case, we might have many free bytes but not be able to satisfy a large request since the free bytes are not contiguous in memory.

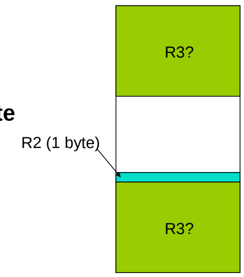
Heap Management

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes



Heap Management

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes



K&R Malloc/Free Implementation

- From Section 8.7 of K&R
  - Code in the book uses some C language features we haven't discussed and is written in a very terse style, don't worry if you can't decipher the code
- Each block of memory is preceded by a header that has two fields:
  size of the block and
  a pointer to the next block
- All free blocks are kept in a linked list, the pointer field is unused in an allocated block

K&R Implementation

- `malloc()` searches the free list for a block that is big enough. If none is found, more memory is requested from the operating system.
- `free()` checks if the blocks adjacent to the freed block are also free
  - If so, adjacent free blocks are merged (coalesced) into a single, larger free block
  - Otherwise, the freed block is just added to the free list

Choosing a block in `malloc()`

- If there are multiple free blocks of memory that are big enough for some request, how do we choose which one to use?
  - best-fit: choose the smallest block that is big enough for the request
  - first-fit: choose the first block we see that is big enough
  - next-fit: like first-fit but remember where we finished searching and resume searching from there

PRS Round 1

•A con of first-fit is that it results in many small blocks at the beginning of the free list

•A con of next-fit is it is slower than first-fit, since it takes longer in steady state to find a match

•A con of best-fit is that it leaves lots of tiny blocks

Tradeoffs of allocation policies

- Best-fit: Tries to limit fragmentation but at the cost of time (must examine all free blocks for each malloc). Leaves lots of small blocks (why?)
- First-fit: Quicker than best-fit (why?) but potentially more fragmentation. Tends to concentrate small blocks at the beginning of the free list (why?)
- Next-fit: Does not concentrate small blocks at front like first-fit, should be faster as a result.

Administrivia

- HW2 Due Wednesday
- HW3 Out Today, Due Sunday
- Proj1 Coming Soon
- If you still aren't enrolled in the course, you may need to talk to Barbara Hightower to get things straightened out. You will almost certainly need to move to section 103.

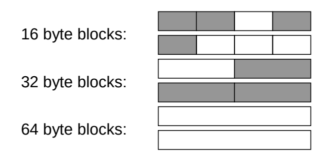
Slab Allocator

- A different approach to memory management (used in GNU `libc`)
- Divide blocks in to "large" and "small" by picking an arbitrary threshold size. Blocks larger than this threshold are managed with a freelist (as before).
- For small blocks, allocate blocks in sizes that are powers of 2
  - e.g., if program wants to allocate 20 bytes, actually give it 32 bytes

Slab Allocator

- Bookkeeping for small blocks is relatively easy: just use a *bitmap* for each range of blocks of the same size
- Allocating is easy and fast: compute the size of the block to allocate and find a free bit in the corresponding bitmap.
- Freeing is also easy and fast: figure out which slab the address belongs to and clear the corresponding bit.

Slab Allocator



16 byte block bitmap: 11011000

32 byte block bitmap: 0111

64 byte block bitmap: 00

Slab Allocator Tradeoffs

- Extremely fast for small blocks.
- Slower for large blocks
  - But presumably the program will take more time to do something with a large block so the overhead is not as critical.
- Minimal space overhead
- No fragmentation (as we defined it before) for small blocks, but still have wasted space!

Internal vs. External Fragmentation

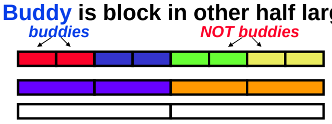
- With the slab allocator, difference between requested size and next power of 2 is wasted
  - e.g., if program wants to allocate 20 bytes and we give it a 32 byte block, 12 bytes are unused.
- We also refer to this as fragmentation, but call it *internal* fragmentation since the wasted space is actually within an allocated block.
- External fragmentation: wasted space between allocated blocks.

Buddy System

- Yet another memory management technique (used in Linux kernel)
- Like GNU's "slab allocator", but only allocate blocks in sizes that are powers of 2 (internal fragmentation is possible)
- Keep separate free lists for each size
  - e.g., separate free lists for 16 byte, 32 byte, 64 byte blocks, etc.

Buddy System

- If no free block of size n is available, find a block of size 2n and split it in to two blocks of size n
- When a block of size n is freed, if its neighbor of size n is also free, coalesce the blocks in to a single block of size 2n
  - **Buddy** is block in other half larger block



- Same speed advantages as slab allocator

Allocation Schemes

- So which memory management scheme (K&R, slab, buddy) is best?
  - There is no single best approach for every application.
  - Different applications have different allocation / deallocation patterns.
  - A scheme that works well for one application may work poorly for another application.

Automatic Memory Management

- Dynamically allocated memory is difficult to track – why not track it **automatically**?
- If we can keep track of what memory is in use, we can reclaim everything else.
  - Unreachable memory is called ***garbage***, the process of reclaiming it is called ***garbage collection***.
- So how do we track what is in use?

Tracking Memory Usage

- Techniques depend heavily on the programming language and rely on help from the compiler.
- Start with all pointers in global variables and local variables (<u>root set</u>).
- Recursively examine dynamically allocated objects we see a pointer to.
  - We can do this in **constant space** by reversing the pointers on the way down
- How do we recursively find pointers in dynamically allocated memory?

Tracking Memory Usage

- Again, it depends heavily on the programming language and compiler.
- Could have only a single type of dynamically allocated object in memory
  - E.g., simple Lisp/Scheme system with only cons cells (61A's Scheme not "simple")
- Could use a ***strongly typed*** language (e.g., Java)
  - Don't allow conversion (casting) between arbitrary types.
  - C/C++ are not strongly typed.
- Here are 3 schemes to collect garbage

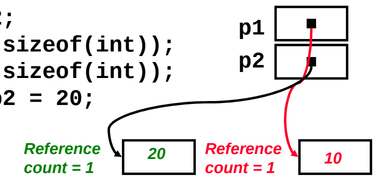
Scheme 1: Reference Counting

- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
- When the count reaches 0, reclaim.
- Simple assignment statements can result in a lot of work, since may update reference counts of many items

Reference Counting Example

- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
  - When the count reaches 0, reclaim.

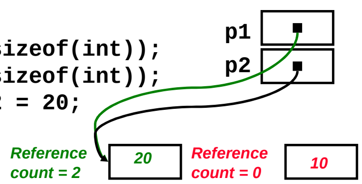
```
int *p1, *p2;
p1 = malloc(sizeof(int));
p2 = malloc(sizeof(int));
*p1 = 10; *p2 = 20;
```



Reference Counting Example

- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
  - When the count reaches 0, reclaim.

```
int *p1, *p2;
p1 = malloc(sizeof(int));
p2 = malloc(sizeof(int));
*p1 = 10; *p2 = 20;
p1 = p2;
```



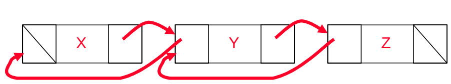
Reference Counting (p1, p2 are pointers)

`p1 = p2;`

- Increment reference count for p2
- If p1 held a valid value, decrement its reference count
- If the reference count for p1 is now 0, reclaim the storage it points to.
  - If the storage pointed to by p1 held other pointers, decrement all of their reference counts, and so on…
- Must also decrement reference count when local variables cease to exist.

Reference Counting Flaws

- Extra overhead added to assignments, as well as ending a block of code.
- Does not work for circular structures!
  - E.g., doubly linked list:



Scheme 2: Mark and Sweep Garbage Col.

- Keep allocating new memory until memory is exhausted, then try to find unused memory.
- Consider objects in heap a graph, chunks of memory (objects) are graph nodes, pointers to memory are graph edges.
  - Edge from A to B => A stores pointer to B
- Can start with the root set, perform a graph traversal, find all usable memory!
- 2 Phases: (1) Mark used nodes;(2) Sweep free ones, returning list of free nodes

Mark and Sweep

- Graph traversal is relatively easy to implement recursively

```
void traverse(struct graph_node *node) {
    /* visit this node */
    foreach child in node->children {
        traverse(child);
    }
}
```

° But with recursion, state is stored on the execution stack.

  ° Garbage collection is invoked when not much memory left

° As before, we could traverse in constant space (by reversing pointers)

Scheme 3: Copying Garbage Collection

- Divide memory into two spaces, only one in use at any time.
- When active space is exhausted, traverse the active space, copying all objects to the other space, then make the new space active and continue.
  - Only reachable objects are copied!
- Use "forwarding pointers" to keep consistency
  - Simple solution to avoiding having to have a table of old and new addresses, and to mark objects already copied (see bonus slides)

PRS Round 2

A. Of {K&R, Slab, Buddy}, there is no best (it depends on the problem).

B. Since automatic garbage collection can occur any time, it is more difficult to measure the execution time of a Java program vs. a C program.

C. We don't have automatic garbage collection in C because of efficiency.

Summary (1/2)

- C has 3 pools of memory
  - Static storage: global variable storage, basically permanent, entire program run
  - The Stack: local variable storage, parameters, return address
  - The Heap (dynamic storage): `malloc()` grabs space from here, `free()` returns it.
- `malloc()` handles free space with freelist. Three different ways to find free space when given a request:
  - First fit (find first one that's free)
  - Next fit (same as first, but remembers where left off)
  - Best fit (finds most "snug" free space)

Summary (2/2)

- Several techniques for managing heap w/ malloc/free: best-, first-, next-fit, slab,buddy
  - 2 types of memory fragmentation: internal & external; all suffer from some kind of frag.
  - Each technique has strengths and weaknesses, none is definitively best
- Automatic memory management relieves programmer from managing memory.
  - All require help from language and compiler
  - Reference Count: not for circular structures
  - Mark and Sweep: complicated and slow, works
  - Copying: move active objects back and forth